
# Implementing Backpropagation

Advanced Concepts of Machine Learning

**I6278383 - Pierre Onghena**
**I6263878 - Simon Hilt**

# 1 Introduction

The software consist out of the four following functions:

```
def sigmoid(z)
def sigmoid_derivative(z)
def training(X, Y)
def predict(Wxh, Why, Bh, By, X)
```

The first step is to create an object of the class ANN() and specify the parameters (bias and weight vectors). To do so, the weight and bias vectors need to be randomly created. In order to retain the trained weight and bias vectors the function *training()* needs to be called with the labeled test set (here the input equals the output). After retrieving the weight and bias vectors, the function *predict()* can be called in order to see the predicted outputs given the trained weights and biases. Also the vector $a^3$ is retrieved to see the confidence of the prediction.

Furthermore, the results from different runs where tracked using Weights and Biases, a tool for experiment tracking. Herewith, hyperparameters as the learning rate and number of epochs are defined to tune upfront. The upshift of this approach was a reduced coordination and experiment tracking overhead while everything was assembled in comprehensive graphs. On the other hand, a well-defined configuration with the tool was needed so that the data being tracked had a clear documentation. For further instructions on how to run the code with or without sweeps, a reference is made to the corresponding readme file.

# 2 Implementation

The neural network is constructed with 8 input nodes, 3 hidden nodes and 8 output nodes. The nodes are connected by weights. For the hidden and output layer an additional bias node is connected. The weight matrix ($\theta^1$) between the the input and output layer is of shape (8,3) and the weight matrix ($\theta^2$) between hidden and output layer is of shape (3,8). The bias for the hidden layer ($b^1$) is of shape (1,3) and the bias ($b^2$) between hidden and output layer is of shape (1,8). The structure of the neural network can be seen in figure 1.

Implemented are two functions and one class. The function *sigmoid(input)* computes the sigmoid function of a value and the function *sigmoid_derivative(input)* computes the output of the derivative sigmoid function. The class *ANN()* contains the two functions *training(X,Y)* and *predict(Wxh, Why, Bh, By, X)*. As the name already suggests, the training function is there in order to train the neural network (retain the weights and biases for the specific prediction task). The number of epochs are the number of rehearsals with which the weights and biases will be adapted and the learning rate is an additional parameter to control the change to the weights and biases for each epoch.

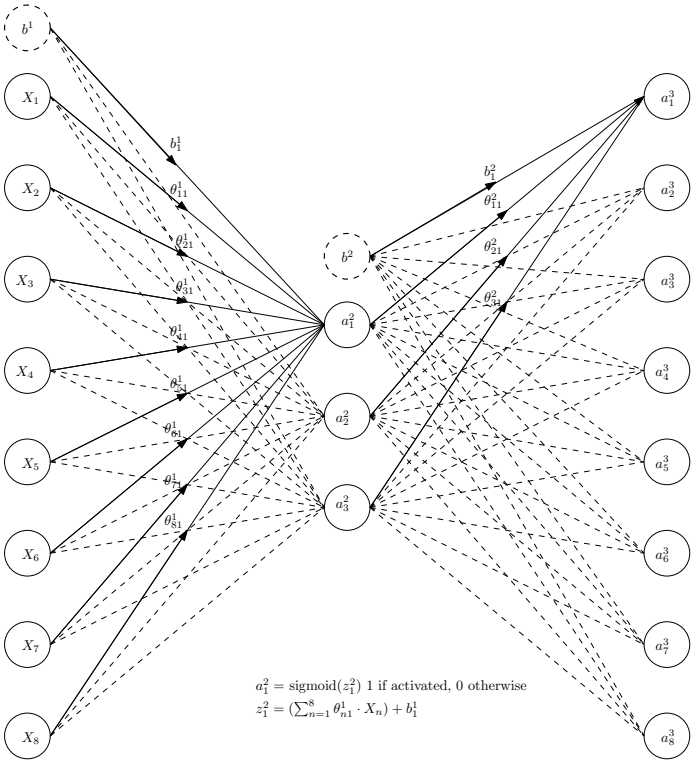


**Figure 1:** Structure of the neural network

# 3 Backpropagation

The most important part of the training function is the backpropagation of the error (adapting the weights/biases based on the costs/loss).
The first step is to compute the deltas ($\delta^3$) at the output nodes. This is done with the formula $\delta_n^3 = a_n^3 - y_n$ where n ranges from 1 to 8 (all output nodes). Based on the costs of each output node, the costs of $\delta^2$ (for the weights of vector $\theta^2$) can be calculated with the following formula:

$$\delta^2 = (\theta^2)^T \delta^3 .* \tag{1}$$

$$\theta_2 = \theta_2 - \alpha \cdot \delta^2 \tag{2}$$

$$\delta^1 = (\theta^1)^T \delta^2 . * g'(z^2) \tag{3}$$

The amount of change applied to the weights and bias can be parametically adjusted with the learning rate and it is automatically adjusted with the gradient descent approach.

The bias for the output layer ($b^2$) is adjusted with respect the costs of the output layer ($b^2 = b^2 - learningRate \cdot \delta^3$). The bias for the hidden layer ($b^1$) will also be similarly adjusted using the gradient of the sigmoid function ($b^1 = b^1 - learningRate \cdot (\delta^2 \cdot g'(z^2))$).

# 4 Hyperparameter tuning

There are multiple ways to find the best hyperparameter combination, in experiments one of the following search methods is usually considered for finding the optimal pair of hyperparameters: grid, random and bayesian search. For this use case, only the grid search will be examined as it computes the outcome of every possible combination described. The corresponding values are predefined in one of the yaml files. While this method searches the feature space exhaustively it also requires a lot of experiments. However, as this assignment entails two parameters to tune, it is considered feasible to execute the grid search in different settings.

## 4.1 Learning rate and Epochs

The first sweep describes the examination of relatively small learning rates ranging from 0.001 to 0.05. In order to test the training accuracy in relation to the number of epochs, the latter has values within the bounds of 100 and 10000. As is illustrated in figure 2, a learning rate of 0.005, 0.01 and 0.25 suggest the need for 5000 or more epochs in order to reach a training accuracy of 100%. On the other hand, a higher learning rate of 0.5 only implies 1000 epochs needed for providing a perfect training accuracy.

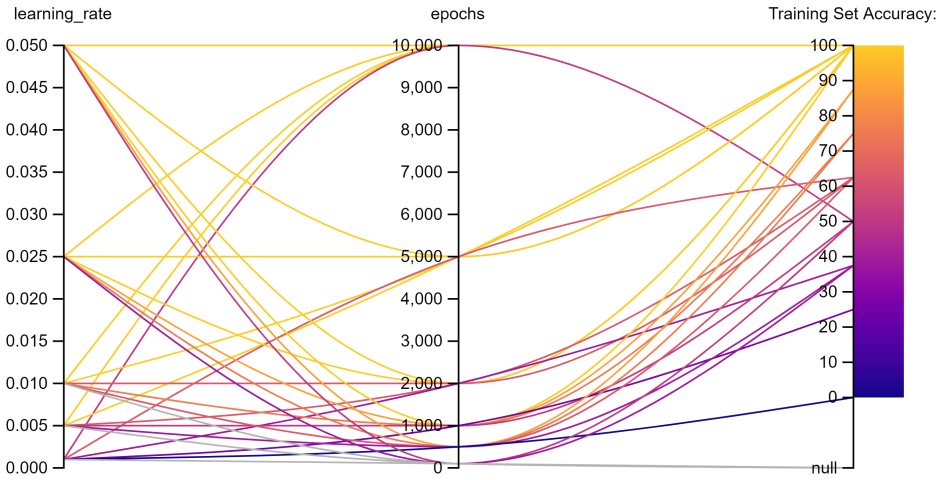


**Figure 2:** Relative low learning rates

Therefore, as can be learned from the previous experiment, the configuration of another sweep can be conducted with higher learning rates to better fit and mimic the uncomplicated problem setting. The learning rates now range from 0.3 to 0.9 and are combined with a lower number of epochs to measure the outcome of the training accuracy. From figure 3 can be derived that iterating over 100 epochs results in an optimal training accuracy in regard to every learning rate. In addition, it can be noticed that the performance for the learning rate 0.9 is unstable and changes distinctively for every number of epochs. Overall, it could be concluded for this problem setting that a learning rate between 0.3 and 0.7 is advisable and results in a lower amount of needed epochs and thus less training to mimic the input in the hidden layer correctly.

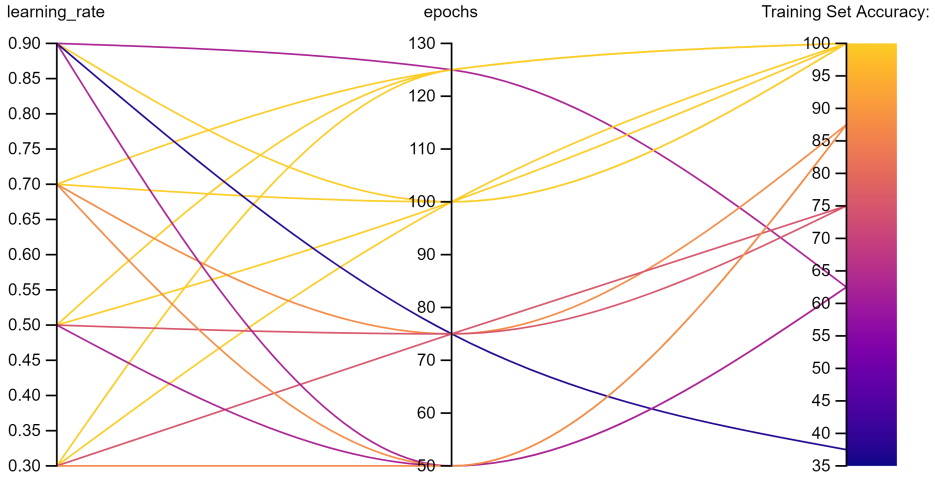


**Figure 3:** High learning rates

## 4.2 Weight and Bias interpretation

With respect to the weights and bias, looking at the vector $a^2$ the activation of each node the hidden layer can be analysed. In order to be able to make distinctions of the input, the hidden layer nodes need to be differently activated based for each different input. It becomes obvious that this is the case. For example for the input [1,0,0,0,0,0,0,0] only the third node in the hidden layer is activated, which then causes only the activation of the first node in the output layer.

In order to make it easier to see the connection of weights and biases, the vectors were tried to be simplified (taking average/relative values). The simplified weights and biases are illustrated at the top of the next page. The retrieved vectors are giving an accuracy of 100%, however the related costs are higher than the weights retrieved directly from the training of the neural network.

The activation of the nodes in the hidden layer is the same as the weights for

that node. This is due to the input, which causes only one input node to be activated for each instance. Therefore also the bias can be neglected.
The network gets more complex with the second weights and bias vector. The calculation for the third layer can be seen in equation 4. It becomes obvious that the bias is needed in this case and also that the weights and bias are in a way composed, such that only one output will be positive depending on which nodes are activated in the hidden layer.
Multiplying the simplified bias and weight vectors with a positive number will result in larger values and therefore also larger difference for the output layer, which is also further reducing the cost. A comparison of the simplified NN and one that is actually trained by the network is given on page 6, each result in an accuracy of 100%.

$$0* \begin{bmatrix} -5 \\ -5 \\ 5 \\ -5 \\ -5 \\ 5 \\ 5 \\ 5 \end{bmatrix} + 0* \begin{bmatrix} -5 \\ -5 \\ 5 \\ 5 \\ 5 \\ -5 \\ -5 \\ 5 \end{bmatrix} + 1* \begin{bmatrix} 5 \\ -5 \\ 5 \\ 5 \\ -5 \\ 5 \\ -5 \\ -5 \end{bmatrix} + \begin{bmatrix} -2.5 \\ 2.5 \\ -12.5 \\ -7.5 \\ -2.5 \\ -7.5 \\ -2.5 \\ -7.5 \end{bmatrix} = \begin{bmatrix} 2.5 \\ -2.5 \\ -7.5 \\ -2.5 \\ -7.5 \\ -2.5 \\ -7.5 \\ -12.5 \end{bmatrix} \quad (4)$$

```
a2:
 [[0. 0. 1.]
  [0. 0. 0.]
  [1. 1. 1.]
  [0. 1. 1.]
  [0. 1. 0.]
  [1. 0. 1.]
  [1. 0. 0.]
  [1. 1. 0.]]

a3:
 [[1. 0. 0. 0. 0. 0. 0. 0.]
  [0. 1. 0. 0. 0. 0. 0. 0.]
  [0. 0. 1. 0. 0. 0. 0. 0.]
  [0. 0. 0. 1. 0. 0. 0. 0.]
  [0. 0. 0. 0. 1. 0. 0. 0.]
  [0. 0. 0. 0. 0. 1. 0. 0.]
  [0. 0. 0. 0. 0. 0. 1. 0.]
  [0. 0. 0. 0. 0. 0. 0. 1.]]
```

```
Simplified bias and weights:
s_b1=  [0, 0, 0]
s_theta1 = [[-5, -5,  5],
            [-5, -5, -5],
            [ 5,  5,  5],
            [-5,  5,  5],
            [-5,  5, -5],
            [ 5, -5,  5],
            [ 5, -5, -5],
            [ 5,  5, -5]]
s_b2 = [ -2.5,  2.5, -12.5, -7.5, -2.5, -7.5, -2.5, -7.5]
s_theta2 = [[-5, -5,  5, -5, -5,  5,  5,  5],
            [-5, -5,  5,  5,  5, -5, -5,  5],
            [ 5, -5,  5,  5, -5,  5, -5, -5]]

Retrieved bias and weights from nn:
b1:
 [-0.47374828 -0.44330602 -0.49944612]
Theta1:
 [[-5.25844516 -5.31910958  3.31533522]
  [-5.09143686 -5.01381376 -5.48046244]
  [ 6.85743294  6.7211964   6.82052939]
  [-6.03262154  5.24289621  5.39924092]
  [-5.07946282  4.11669373 -4.99408253]
  [ 4.9689682  -6.06131745  5.43755129]
  [ 3.78014311 -5.35803243 -4.92205818]
  [ 5.41085372  5.27368885 -6.04221909]]
b2:
 [ -7.87191512   7.37285297 -34.23729987 -21.43018119  -8.01692089
 -21.3683717   -7.9756832  -21.66416788]
Theta2:
 [[-16.88616624 -16.26720537  13.5974542  -15.8073766  -16.33615775
 14.17440653  16.01330057  14.57241913]
 [-16.82639622 -16.08679542  13.55759196  14.10931159  15.90499392
 -15.86534944 -16.51598387  14.38181734]
 [ 16.22417328 -16.6452962   13.59938296  14.62926368 -16.33317598
 14.54878863 -16.41888777 -15.79674816]]
```